MASSACHUSETTS INSTITUTE OF TECHNOLOGY
6.001 -- Structure and Interpretation of Computer Programs
Fall 1997

Problem Set 6 Solutions

Tutorial Exercise 4 (SICP 3.23)

Here is an implementation of dequeue using doubly-linked lists. Dequeue is represented as a cons cell, with the car pointing to the front of the element list and the cdr pointing to the rear of the list. Thus,

```
(define front-ptr car)
(define rear-ptr cdr)
```

Each element in the dequeue has *two* cons cells: the car of the first cons cell points to the second cons cell, and the cdr of the first cons cell points to the next element (successor) in the list. The car of the second cons cell points to the stored item and the cdr of the second cons cell points to the previous element (predecessor) in the list.

```
(define (make-dequeue) (cons nil nil))
(define (empty-dequeue? dq)
 (and (pair? dq)
      (null? (front-ptr dq))
      (null? (rear-ptr  dq))))
(define (front-insert-dequeue! dq item)
  (let ((new (cons (cons item nil) (front-ptr dq))))
    (if (empty-dequeu? dq)
        (set-rear-ptr! dq new)
        (set-cdr! (car (front-ptr dq)) new))
    (set-front-ptr! dq new)))
(define (rear-insert-dequeue! dq item)
  (let ((new (cons (cons item (rear-ptr dq)) nil)))
    (if (empty-dequeu? dq)
        (set-front-ptr! dq new)
        (set-cdr! (rear-ptr dq) new))
    (set-rear-ptr! dq new)))
(define (front-delete-dequeue! dq)
  (if (empty-dequeue? dq)
      (error "Cannot delete from an empty queue: FRONT-DELETE-DEQUEUE!")
      (if (null? (cdar (rear-ptr dq)))
           ;;; delete from a one element dequeue
           (begin
             (set-front-ptr! dq nil)
             (set-rear-ptr!  dq nil))
           ;;; delete from a dequeue with at least two elements
           (begin
             (set-front-ptr! dq (cdr (front-ptr dq)))
             (set-cdr! (car (front-ptr dq)) nil)))))
(define (rear-delete-dequeue! dq)
  (if (empty-dequeue? dq)
```

```
          (error "Cannot delete from an empty queue: FRONT-DELETE-DEQUEUE!")
          (if (null? (cdar (rear-ptr dq)))
               ;;; delete from a one element dequeue
               (begin
                 (set-front-ptr! dq nil)
                 (set-rear-ptr!  dq nil))
               ;;; delete from a dequeue with at least two elements
               (begin
                 (set-rear-ptr! dq (cdar (rear-ptr dq)))
                 (set-cdr! (rear-ptr dq) nil)))))
```

There is an amortized cost theta(1) solution (i.e. the time to service any sequence of length k of requests of inserts and deletes is theta(k)) that does not use a cyclic list structure. The idea is to maintain two (singly-linked) lists of elements, l1 and l2, and shuffle elements from one list to the other only iff

abs(length(l1)- length(l2)) > n/2, where n is the number of requests.

Pre-Lab Exercise 1

```
(depth-first-tree-search 10 tree)
```

The order is: 1 2 4 5 8 9 3 6 7

```
(other-tree-search 10 tree)
```

The order is: 1 2 4 5 8 9 3 6 7

Pre-Lab Exercise 2

```
(breadth-first-tree-search 10 tree)
```

The order is: 1 2 3 4 5 6 7 8 9

Lab Exercise 3

Add the nodes visited into the `nodes` list while performing the `same-label?` Operation. `clear-visited-lables!` reinitialize the nodes list to null. The order of the nodes visited is the reverse order of the nodes list, since the latest node is at the head of the list.

```
(define (install-labels!)
  (let ((nodes ‘()))
    (set! same-label?
          (lambda (given-label node-label)
            (begin
              (set! nodes (cons node-label nodes))
              (equal? given-label node-label))))
    (set! clear-visited-labels!
          (lambda () (set! nodes ‘())))
    (set! show-visited-labels! (lambda () (reverse nodes)))))
```

Install the new set of procedures by:

```
(install-labels!)

(depth-first-tree-search 10 tree)
(show-visited-labels!)
;Value: (1 2 4 5 8 9 3 6 7)
(clear-visited-labels!)

(other-tree-search 10 tree)
(show-visited-labels!)
;Value: (4 8 9 5 2 6 7 3 1)
(clear-visited-labels!)

(breadth-first-tree-search 10 tree)
(show-visited-labels!)
;Value: (1 2 3 4 5 6 7 8 9)
(clear-visited-labels!)
```

Lab Exercise 4

First, lets define the tree given in PreLab Exercise 1 as an adjacency list.

```
(define adjacency-list-tree
  ‘((5 (8 9))
    (4 ())
    (2 (4 5))
    (1 (2 3))
    (6 ())
    (3 (6 7))
    (8 ())
    (9 ())
    (7 ())))
```

We can use the `link-up` routine given in *search.scm* to create a tree structure out of the adjacency list representation. The procedure `set-up-list` will convert an adjacency list to a linked graph and return the root element. However, we need to find the root node after creating the graph representation of the tree. The `find-root` routine will mark the nodes that are children of a node. Thus, the only unmarked node should be the root node.

```
(define (find-root graph)
  (begin
    (map (lambda (e)
            (map (lambda (a) (if a (mark! a))) (adjacent-nodes e)))
         graph)
    (define ret (car (filter (lambda (e) (not (marked? e))) graph)))
    (install-marking-package! graph)
    ret))

(define (setup-list adjacency-list)
  (let ((linked-list (linkup adjacency-list)))
    (begin
      (install-value-package! linked-list)
      (find-root linked-list))))
```

We create the utility function `apply-or,` which applies a function to each element of the list until the function evaluates to true. And we redefine `label-of` and `sons-list` to the corresponding routines provided in `install-value-package!`

```
(define (apply-or lst op)
  (if (null? lst)
      #f
      (or (op (car lst))
          (apply-or (cdr lst) op))))
(define empty? null?)
(define same-label? =)
(define label-of node-name)
(define sons-list adjacent-nodes)
```

Now, the change required in the search algorithms are trivial. Instead of only testing left and right subtrees, we use `apply-or` to test all the sub-trees of an element. In `breadth-first-tree-search`, we append the entire list of subtrees instead of the two subtrees in the binary tree algorithm.

```
(define (depth-first-tree-search label tree)
  (define (do-dfs tree)
    (and (not (empty? tree))
         (or (same-label? label (label-of tree))
             (apply-or (sons-list tree) do-dfs))))
  (do-dfs tree))

(define (other-tree-search label tree)
  (define (do-ots tree)
    (and (not (empty? tree))
         (or (apply-or (sons-list tree) do-ots)
             (same-label? label (label-of tree)))))
  (do-ots tree))

(define (breadth-first-tree-search label tree)
  (define (breadth-first-trees subtrees)
    (and (not (null? subtrees))
         (let ((subtree (car subtrees)))
           (cond ((empty? subtree) (breadth-first-trees (cdr subtrees)))
                 ((same-label? label (label-of subtree)) #t)
                 (else (breadth-first-trees
                         (append (cdr subtrees)
                                 (sons-list subtree))))))))
  (breadth-first-trees (list tree)))
```

Here is a useful routine that will print just the names of the tree nodes; printing the tree structure itself can be very messy.

```
(define (print-tree tree)
  (if tree
      (cons (node-name tree) (map print-tree (sons-list tree)))))
```

Here is the sequence of commands used to test the new search routines. We first test this on the same binary tree, but in the new representation.

```
(install-labels!)

(define tree (setup-list adjacency-list-tree))

(depth-first-tree-search 10 tree)
(show-visited-labels!)
;Value: (1 2 4 5 8 9 3 6 7)
(clear-visited-labels!)

(other-tree-search 10 tree)
(show-visited-labels!)
;Value: (1 3 7 6 2 5 9 8 4)
(clear-visited-labels!)

(breadth-first-tree-search 10 tree)
(show-visited-labels!)
;Value: (1 2 3 4 5 6 7 8 9)
(clear-visited-labels!)
```

As expected, the answers were the same. Now, lets create a more complicated tree using the following adjacency list.

```
(define adjacency-list-tree2
  ‘((root (a b c))
    (a (a1 a2 a3))
    (b (b1 b2))
    (c (c1  c2 c3 c4))
    (a1 ())
    (a2 (a2i))
    (a3 (a3i a3j))
    (b1 (b1i b1j b1k))
    (b2 (b2i b2j))
    (c1 (c1i))
    (c2 ())
    (c3 (c3i c3j))
    (c4 ())
    (a2i ())
    (a3i ())
    (a3j (a3jA a3jB))
    (b1i ())
    (b1j ())
    (b1k ())
    (b2i ())
    (b2j ())
    (c1i (c1iA))
    (a3jA ())
    (a3jB ())
    (c1iA (c1iAx c1iAy))
    (c1iAx ())
    (c1iAy ())))

(define tree2 (setup-list  adjacency-list-tree2))
```

Here is the tree structure

```
(root (c (c4)
         (c3 (c3j)
             (c3i))
         (c2)
         (c1 (c1i (c1ia (c1iax)
                        (c1iay)))))
      (b (b2 (b2j)
             (b2i))
         (b1 (b1k)
             (b1j)
             (b1i)))
      (a (a3 (a3j (a3jb)
                  (a3ja))
             (a3i))
         (a2 (a2i))
         (a1)))
```

In the first set of tests, look for an element that is not in the tree.

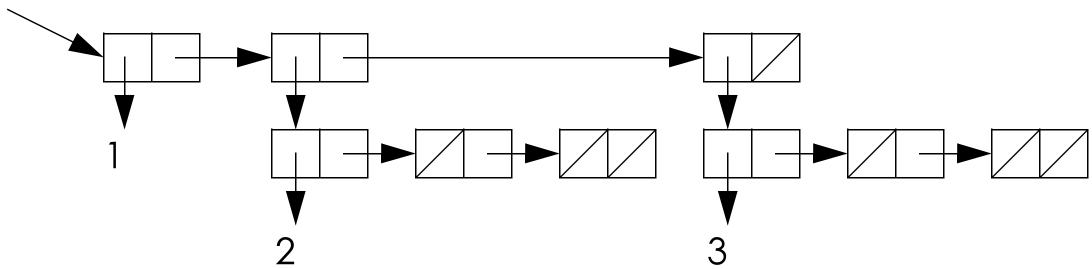
```
(depth-first-tree-search ‘b1b tree2)
(show-visited-labels!)
;Value: (root a a1 a2 a2i a3 a3i a3j a3ja a3jb b b1 b1i b1j b1k b2 b2i b2j c
c1 c1i c1ia c1iax c1iay c2 c3 c4)
(clear-visited-labels!)

(other-tree-search ‘b1b tree2)
(show-visited-labels!)
;Value: (a1 a2i a2 a3i a3ja a3jb a3j a3 a b1i b1j b1k b1 b2i b2j b2 b c1iax
c1iay c1ia c1i c1 c2 c3 c4 c root)
(clear-visited-labels!)

(breadth-first-tree-search ‘b1b tree2)
(show-visited-labels!)
;Value: (root a b c a1 a2 a3 b1 b2 c1 c2 c3 c4 a2i a3i a3j b1i b1j b1k b2i
b2j c1i a3ja a3jb c1ia c1iax c1iay)
(clear-visited-labels!)
```

Now, lets find an element in the tree.

```
(depth-first-tree-search ‘b1j tree2)
(show-visited-labels!)
;Value: (root a a1 a2 a2i a3 a3i a3j a3ja a3jb b b1 b1i b1j)
(clear-visited-labels!)

(other-tree-search ‘b1j tree2)
(show-visited-labels!)
;Value: (a1 a2i a2 a3i a3ja a3jb a3j a3 a b1i b1j)
(clear-visited-labels!)

(breadth-first-tree-search ‘b1j tree2)
(show-visited-labels!)
;Value: (root a b c a1 a2 a3 b1 b2 c1 c2 c3 c4 a2i a3i a3j b1i b1j)
(clear-visited-labels!)
```

Pre-Lab Exercise 4

```
(define tree '(1 (2 () ()) (3 () ())))
;Value: "tree --> (1 (2 #f #f) (3 #f #f))"
```




```
(set-car! (cdr tree) tree)
;Value: #[unspecified-return-value]
```

```
(caddr tree)
;Value: (3 #f #f)

(car (cddadr tree))
;Value: (3 #f #f)

(breadth-first-tree-search 4 tree)
```

This will run forever because of the circular nature of the structure.

Lab Exercise 5

Lets check for reporting paths between Saman and Arjuna and Tara and Jill. First, we need to create the linked graph, and then install the marking and value packages.

```
(define 6001-graph (linkup 6001-reporting))

(install-marking-package! 6001-graph)
(install-value-package! 6001-graph)

(define (found-it? node)
  (equal? (node-name node) 'Arjuna))

(depth-first-search 'Saman 6001-graph found-it?)
;Value: arjuna

(breadth-first-search 'Saman 6001-graph found-it?)
;Value: arjuna
```

```
(define (found-it? node)
  (equal? (node-name node) ‘Jill))

(depth-first-search ‘Tara 6001-graph found-it?)
;Value: jill

(breadth-first-search ‘Tara 6001-graph found-it?)
;Value: jill
```

Lab Exercise 6

The procedure `check` will set up the label collection routines, define the `fond-it?` routine and run the given search algorithm.

```
(define (check search graph from to)
  (let ((nodes ‘())
         (num-nodes 0))
    (begin
      (define clear-visited-labels!
         (lambda ()
             (set! nodes ‘())
             (set! num-nodes 0)))
      (define show-visited-labels! (lambda () (reverse nodes)))
      (define found-it?
         (lambda (nd)
             (begin
                (set! nodes (cons (node-name nd) nodes))
                (set! num-nodes (+ num-nodes 1))
                (if (= num-nodes 10)
                    #t
                   (equal? (node-name nd) to)))))
      (clear-visited-labels!)
      (search from graph found-it?)
      (show-visited-labels!))))
```

The first test will run to completion, while the second one will return after visiting 10 nodes.

```
(check depth-first-search 6001-graph ‘Tara ‘Arjuna)
;Value: (tara saman martin albert rod daniel jill david jeremy arjuna)

(check breadth-first-search 6001-graph ‘Tara ‘Arjuna)
;Value: (tara saman martin albert rod daniel jill jeremy ben david)
```

Lab Exercise 7

The procedure `check-path` will define the `fond-it?` routine and run the given search algorithm and return the paths.

```
(define (check-path search graph from to)
    (begin
      (define found-it?
         (lambda (nd)
            (equal? (node-name nd) to)))
      (search from graph found-it?)
      (path-back-from to graph))))
```

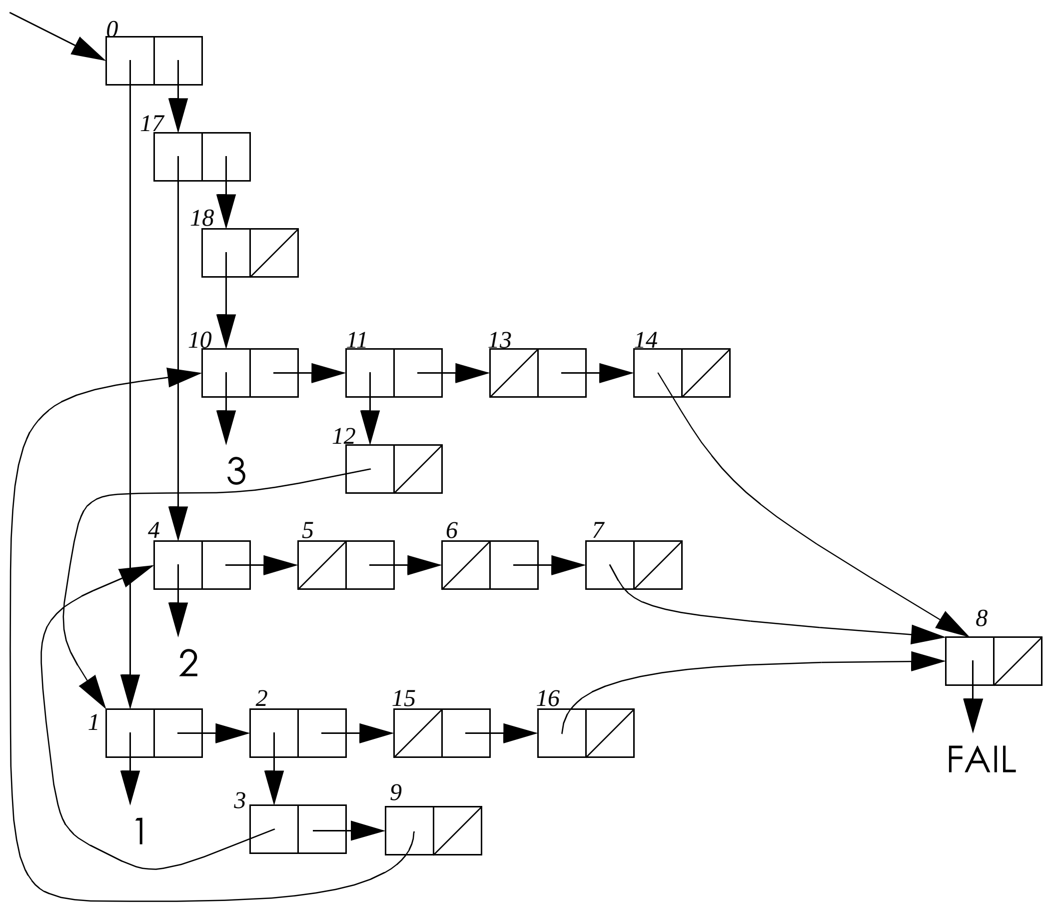
We run the four tests performed in Exercise 5.

```
(check-path depth-first-search-path-back 6001-graph ‘Saman ‘Arjuna)
;Value: (arjuna jeremy albert martin saman)

(check-path breadth-first-search-path-back 6001-graph ‘Saman ‘Arjuna)
;Value: (arjuna jeremy albert saman)

(check-path depth-first-search-path-back 6001-graph ‘Tara ‘Jill)
;Value: (jill rod albert martin saman tara)

(check-path breadth-first-search-path-back 6001-graph ‘Tara ‘Jill)
;Value: (jill albert saman tara)
```

Pre-Lab Exercise 8a

```
(define ex8-graph (linkup ‘((1 (2 3)) (2 ()) (3 (1)))))
```

Here is the box and pointer diagram for the linked-up graph.



The italic numbers are the cons-cell identification numbers calculated in the next problem.

Pre-Lab Exercise 8b

Lets write a function to help identify the cons-cell structure. The following routine creates a shadow structure that keeps tab of each cons-cell in the cyclic graph. `lst` list has an entry for each cons-cell and a number associated the cell. The function `find` will traverse the cyclic structure and add entries to the list. Then, we will print the number associated with each cons-cell and information on it's car and cdr. If either car or cdr points to a cons-cell, the number associated with that cons-cell will be printed. If car or cdr points to a value, the value will be printed.

```
(define (cyclic-print cyc)
  (let ((lst '())
        (cnum 0))
    (begin
      (define (find c)
         (cond ((not (pair? c)) #t)
               ((assq c lst) #t)
               (else (begin
                        (set! lst (cons (list c cnum) lst))
                        (set! cnum (inc cnum))
                        (find (car c))
                        (find (cdr c))))))
      (define (prstr e)
         (if (assq e lst)
             (cadr (assq e lst))
             (list 'Value: e)))
      (find cyc)
      (map (lambda (e)
               (list (cadr e) (prstr (caar e)) (prstr (cdar e))))
       (reverse lst)))))

(cyclic-print ex8-graph)
```

```
((0 1 17)
 (1 (value: 1) 2)
 (2 3 15)
 (3 4 9)
 (4 (value: 2) 5)
 (5 (value: #f) 6)
 (6 (value: #f) 7)
 (7 8 (value: #f))
 (8 (value: "FAIL") (value: #f))
 (9 10 (value: #f))
 (10 (value: 3) 11)
 (11 12 13)
 (12 1 (value: #f))
 (13 (value: #f) 14)
 (14 8 (value: #f))
 (15 (value: #f) 16)
 (16 8 (value: #f))
 (17 4 18)
 (18 10 (value: #f)))
```

In fact, you can verify that this is the same structure as the box and pointer diagram in the previous section. Italic numbers in that diagram represent the numbers assigned to the cons-cell.

Lab Exercise 9

The following procedure will visit each node of the graph and create the adjacency list for that node.

```
(define (unlink graph)
  (install-linked-value-package! graph)
  (map (lambda (ex) (list (node-name ex)
       (map node-name (adjacent-nodes ex))))
       graph))
```

We check the unlink routine by:

```
(define adj-list '((1 (2 3)) (2 ()) (3 (1))))

(equal? (unlink (linkup adj-list)) adj-list)
;Value: #t

(equal? (unlink (linkup 6001-reporting)) 6001-reporting)
;Value: #t
```